CS 4530: Fundamentals of Software Engineering
Module 11.2: Application-Level Patterns

Adeel Bhutta, Rob Simmons and Mitch Wand

Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this module, you should be able to:
  - describe the basic ideas of the following architectures, with examples and pictures
    - anarchic
    - layered
    - pipeline
    - event-driven
    - Microkernel (plug-in)

Three Scales of Design

The Structural Scale

- key questions: what are the pieces? how do they fit together to form a coherent whole?

The Interaction Scale

- key questions: how do the pieces interact? how are they related?

The Code Scale

- key question: how can I make the actual code easy to test, understand, and modify?

Design at larger scales

- Metaphor: building architecture
- How do the pieces fit together? Are there parts we can reuse?
- Will the result be structurally sound? earthquake-resistant? economical to build? easy to maintain?

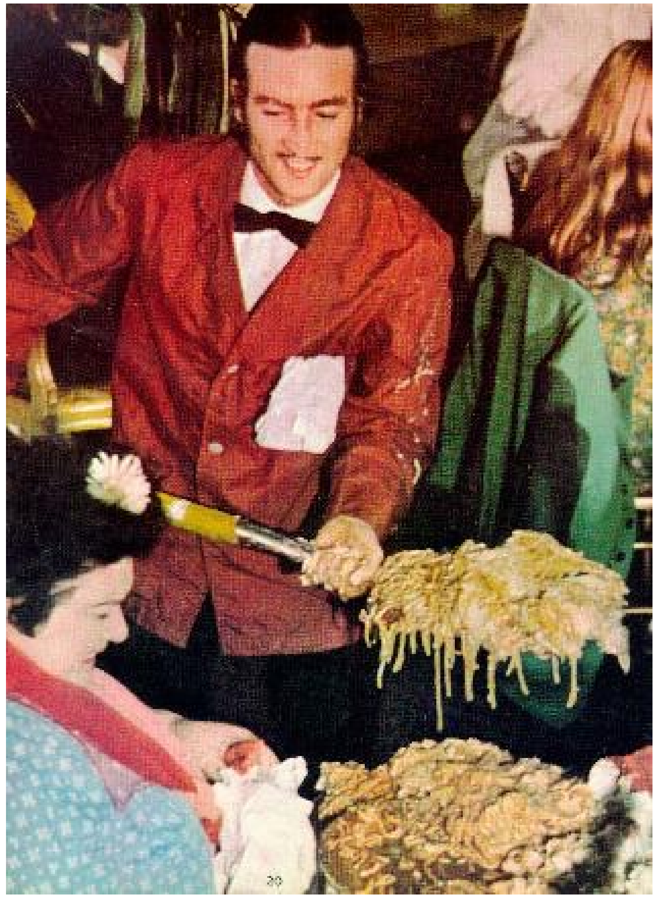
Goal: Create a high-level picture of the system

- Abstract details away into reusable components
- Allows for analysis of high-level design before implementation
- Enables exploration of design alternatives
- Reduce risks associated with building the software

Architecture #0: Anarchic

- A single app, with no particular organization
- Also known as: "spaghetti code"
- May still have useful interfaces for some degree of encapsulation and modularity.
  - but is there a method to the madness?

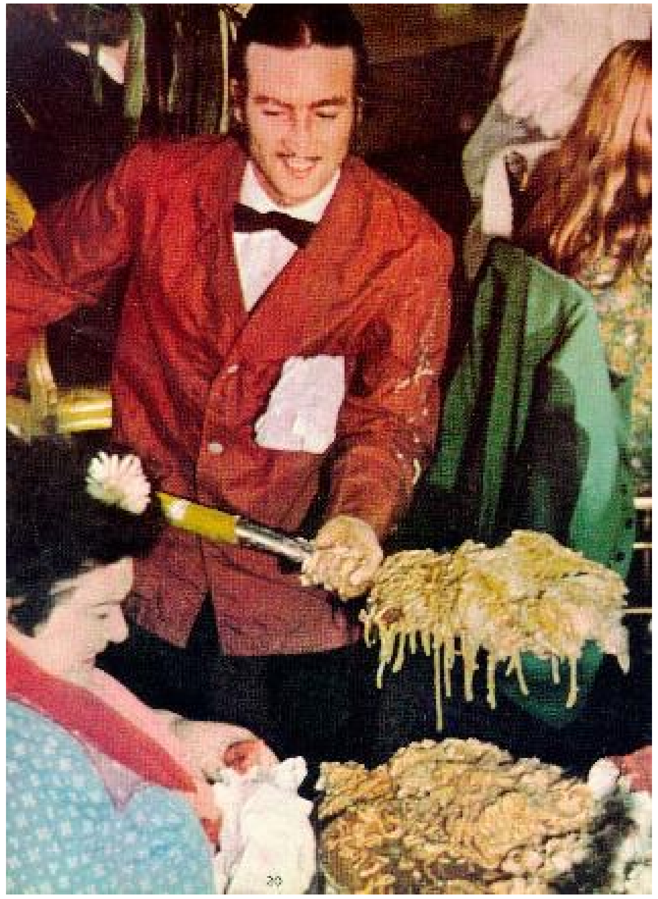
Shakespeare, *Hamlet*. The exact quote is: "Though this be madness, yet there is method in't" (Polonius, Act 2, Scene 2)



Brian Foote and Joe Yoder

Architecture #0: Anarchic

- OK for single-developer, short-lived projects
- But
  - what happens if you want to add a new developer
  - what happens if you need to come back to the code later?



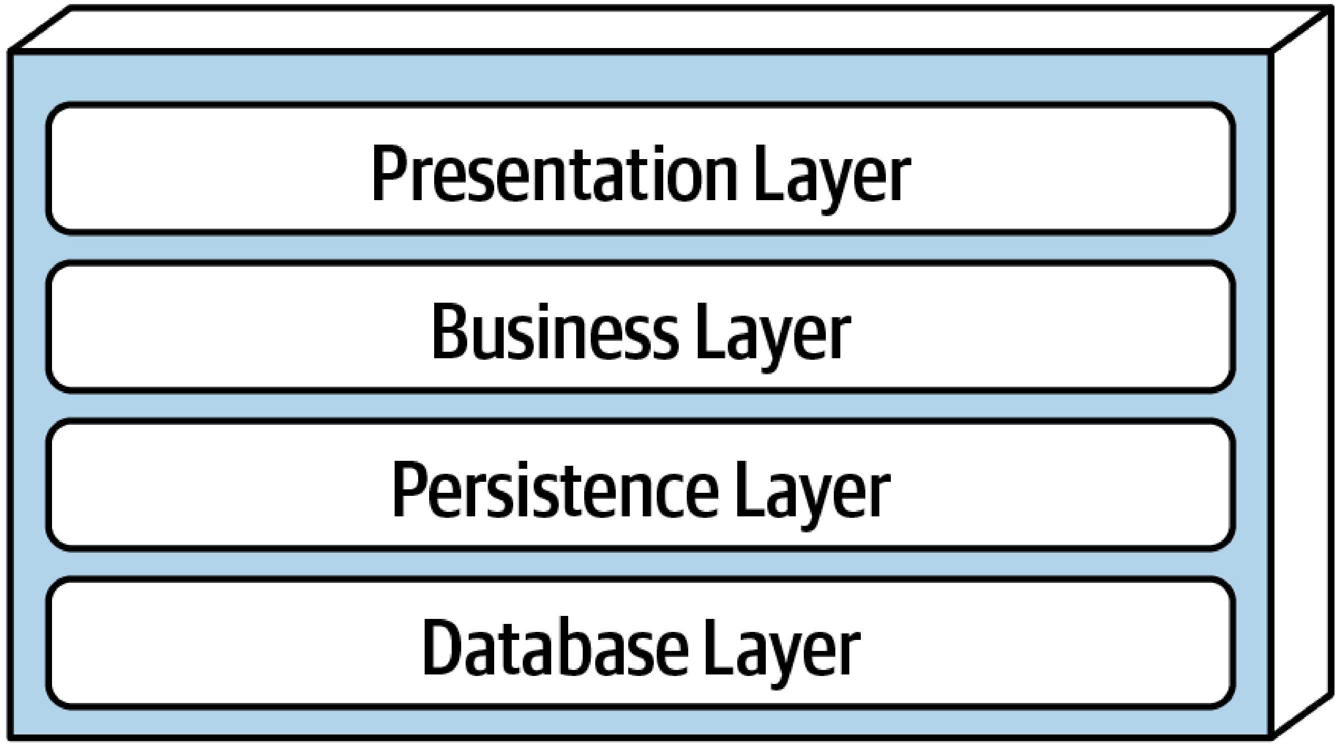
Brian Foote and Joe Yoder

Metaprinciple: Modules and Interfaces

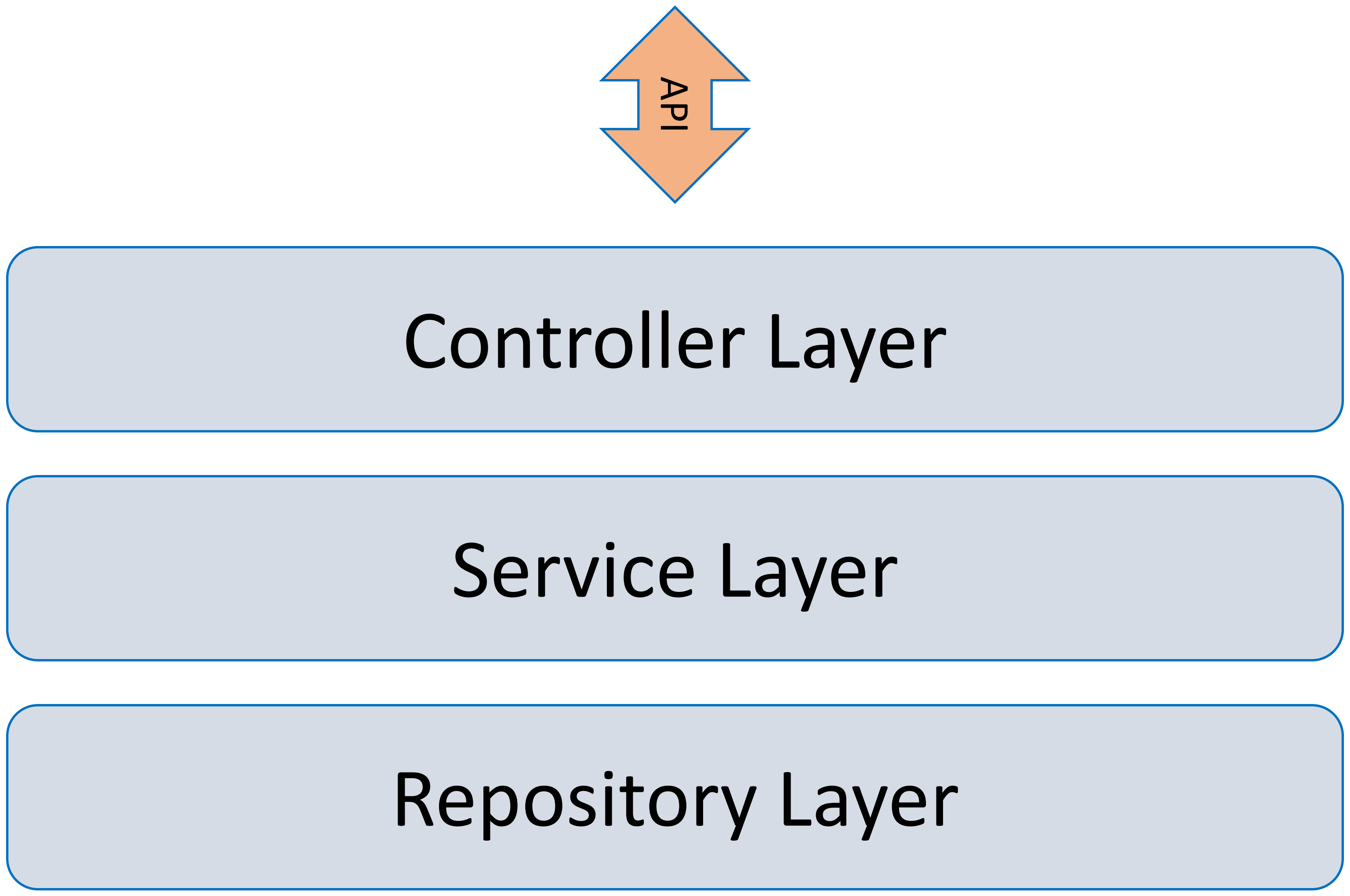
- Our system will be made up of *modules*.
- Each module will have one or more *clients* that utilize its services.
- Each module offers a well-defined *interface* that the clients use when they want to use the services of the module
- The modules may be organized in different ways; that's the main topic of this lesson
- In practice, modules may or may not be organized as neatly as this slide suggests.

Architecture #1: Layered

- Each layer has specific responsibility
- Each layer depends on services from the layer or layers below
- Organize teams by Layer
    - different layers require different expertise
- When the layers are run on separate pieces of hardware, they are sometimes called "tiers"



Controller-Service-Repository



Layered Architecture (contd)

- Typical organization for operating systems
- Layers communicate through procedure calls and callbacks ("up-calls")
- Well-defined interfaces are a must!

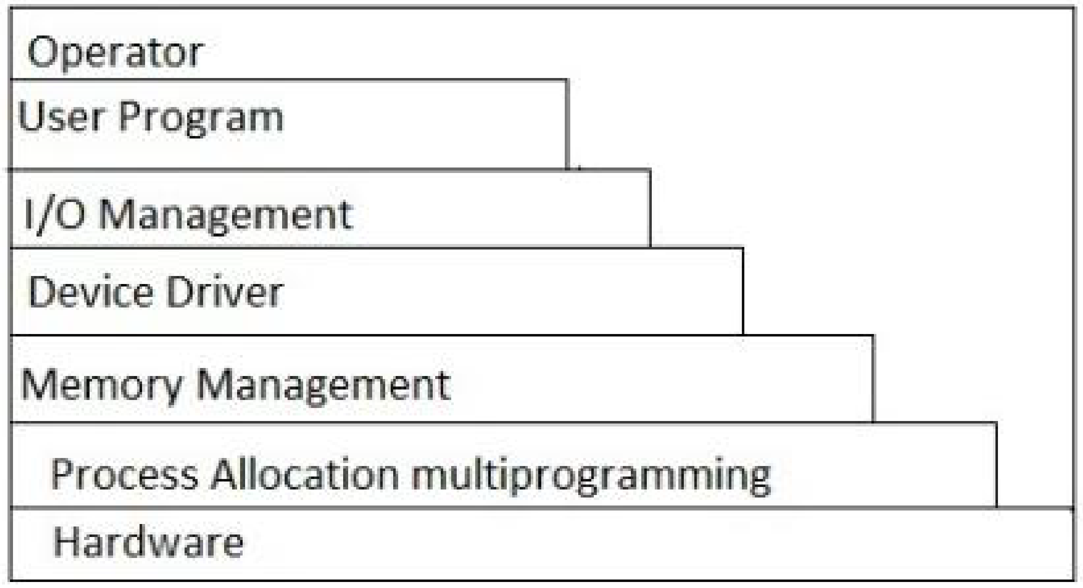
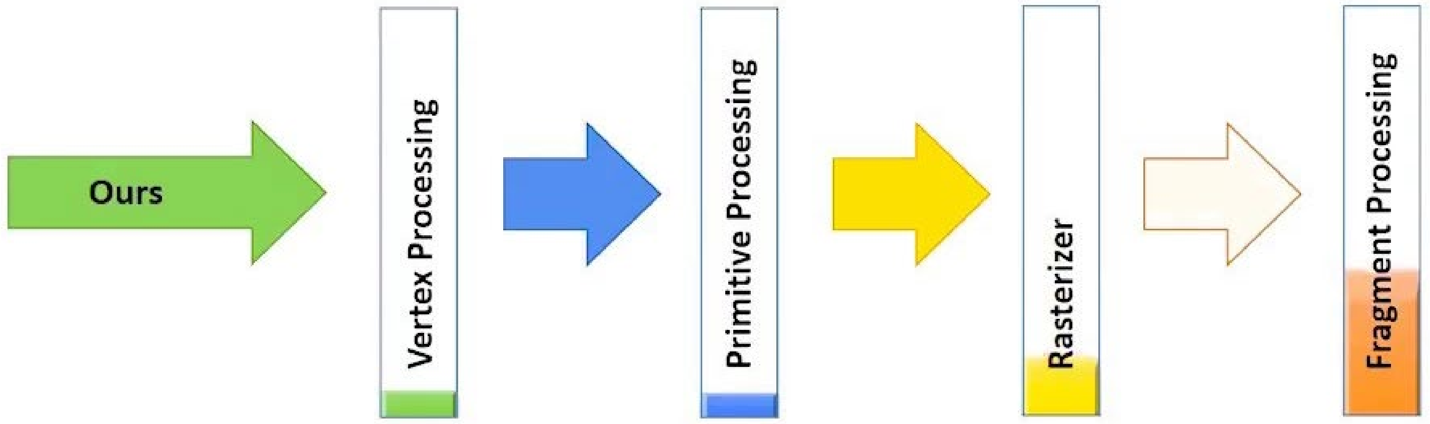
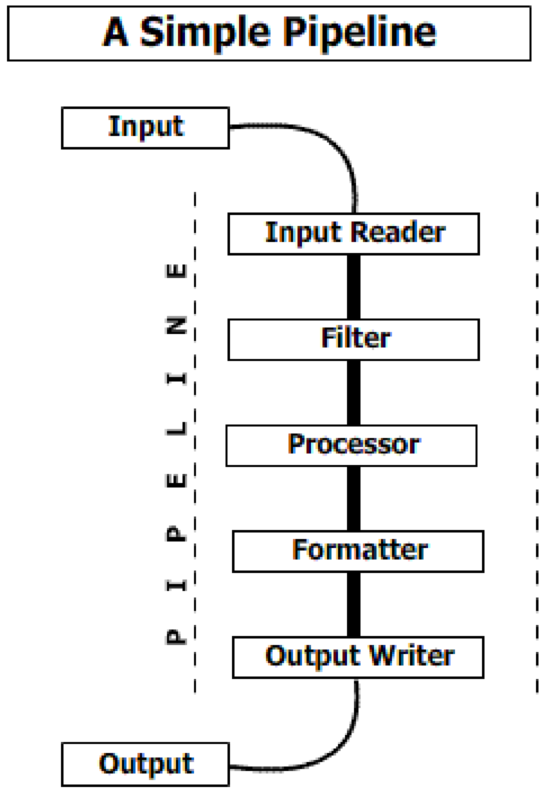


fig:- layered Architecture

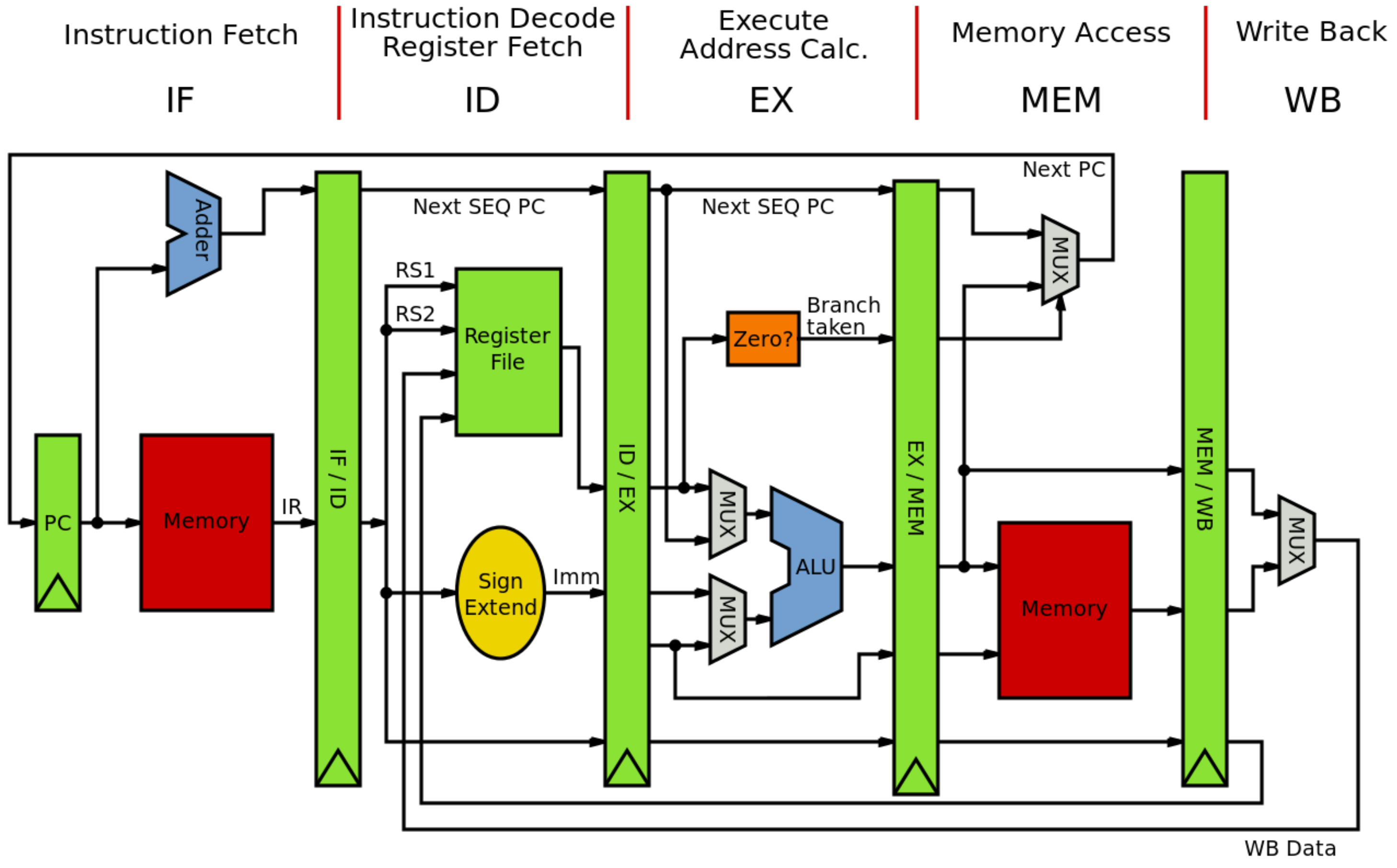
Architecture #2: Pipeline

- Modules are arranged in order of processing.
- Good for complex straight-line processes, eg image processing





Also good for visualizing hardware



How do the stages communicate?

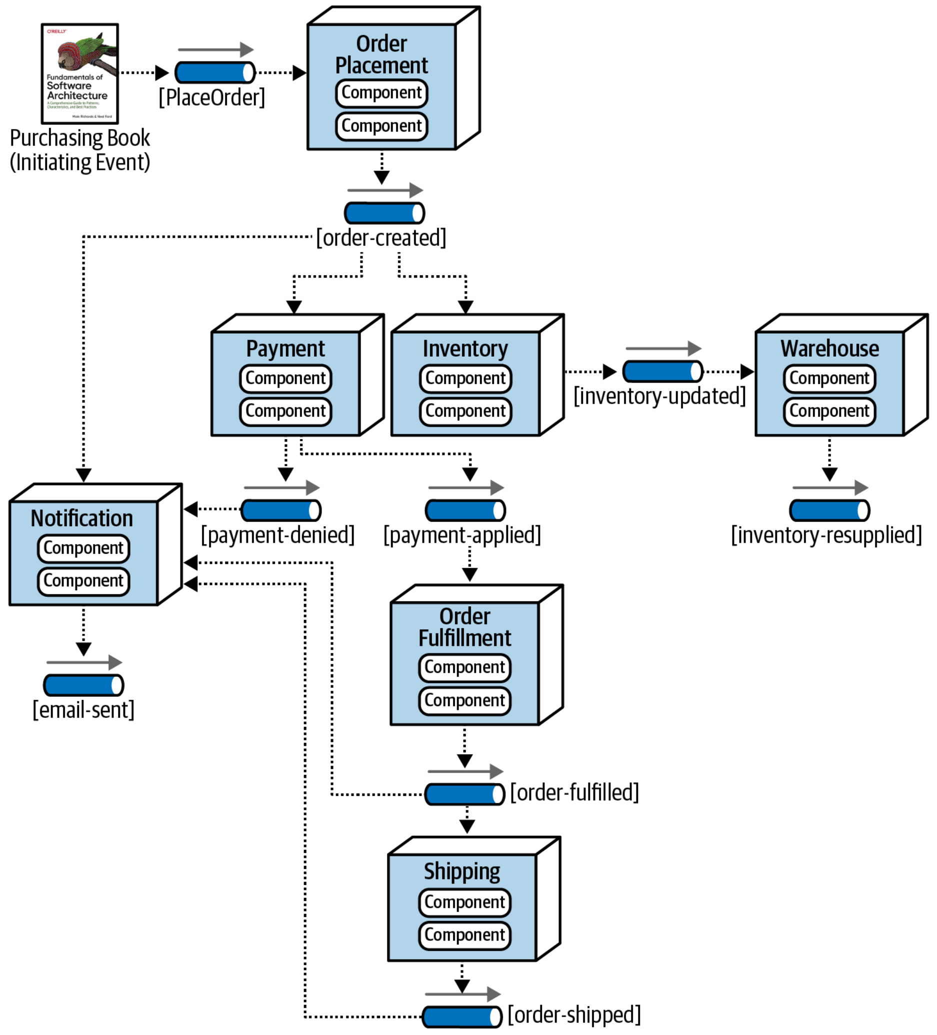
- That's the next-level decision
  - data-push (each stage invokes the next)
  - demand-pull (each stage demands data from its predecessor)
  - queues? buffers?
  - ??
- That's what we talked about in Lesson 11.1

In Express, each stage gets an object that represents the rest of the pipeline

```
app.use((req, res) => {
  res.status(404).json({
    error: 'Not Found',
    message:
      `Route ${req.method} ${req.originalUrl} not found`
  });
```

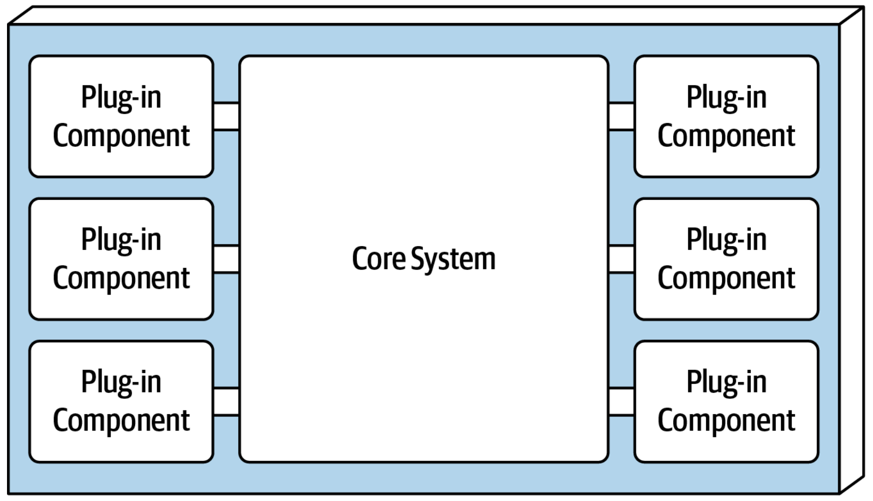
Architecture #3: Event-Driven Architecture

- Modules are organized by client's workflow
- Each processing unit has an in-box and one or more out-boxes
- Each unit takes a task from its inbox, processes it, and puts the results in one or more outboxes.
- Stages may be connected by asynchronous message queues
- Or use the observer pattern, where each unit observes changes in its upstream units.
- Conditional flow



Architecture #4: Plugins ("microkernel")

- System consists of a small core (the "microkernel") for essential functions, and lots of hooks for adding other services
- Highly extensible
- Plug-ins can be designed by small, less-experienced teams– even by users!
- Connection methods may vary
    - often: core provides default behaviors that are overridable



Key Concepts for Plugin Architecture

- **Activation Events**: when does your extension run?
- **Host API**: what procedures in the host app can your extension call?
- **Contribution Point**: what your extension contributes to the host (e.g. new commands, menus, pipeline stages, etc.)

Example 1: git hooks

- git provides a fixed set of activation events (files in .git/hooks)
- the user can extend git's default behavior by changing these files

```
$ cat .git/hooks/pre-merge-commit.sample
#!/bin/sh
#
# An example hook script to verify what is about to be committed.
# Called by "git merge" with no arguments.  The hook should
# exit with non-zero status after issuing an appropriate message to
# stderr if it wants to stop the merge commit.
#
# To enable this hook, rename this file to "pre-merge-commit".

. git-sh-setup
test -x "$GIT_DIR/hooks/pre-commit" &&
        exec "$GIT_DIR/hooks/pre-commit"
:
```

Example 2: express

```
export const createApp = (): express.Application => {
  const app = express();

  // Middleware for parsing JSON requests
  app.use(express.json());

  // Addition endpoint
  app.get('/sum/:i/:j', getSum);

  // get the rest of the routes from frontend/dist
  app.use(express.static('frontend/dist'));

  app.use((req, res) => {
    res.status(404).json({
      error: 'Not Found',
      message: `Route ${req.method} ${req.originalUrl} not found`
    });
  });
```

Review

- You should now be able to:
  - describe the basic ideas of the following architectures, with examples and pictures
    - anarchic
    - layered
    - pipeline
    - event-driven
    - microkernel